Chapter 2

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly*, *Diversification* and *Runtime Randomization*. We present a summary of the relevant related work and the key concepts and background that we use along with this writing. We select the discussed works by their novelty and critical insights to provide automatic diversification.

In Section 2.1 we describe the context in which this dissertation is based, WebAssembly. We include both usage scenarios for Wasm and its main security issues. In Section 2.2 we discuss the main diversification techniques in the wild, we introduce the concept of superoptimization used in this work, and we end the section by highlighting superdiversification as the cornerstone of our contributions are based. In Section 2.3 we mention close works to runtime diversification and how diversification can be used to construct resilient binaries.

2.1 WebAssembly

In this section, we introduce an overview of the motivation for WebAssembly and its usage. Besides, we describe the process to obtain Wasm programs and how this novel technology evolves from being only-browser-based to standalone executions in the backend. Nevertheless, we describe its significant limitations regarding security, that is our main motivation for our research.

The WebAssembly (Wasm) language was first publicly announced in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine. It is designed to address the problem of safe, fast, portable, and compact low-level code on the Web. A paper by Haas et al. [24] formalizes the language and its type system. Since 2015, major web browsers have implemented support for the standard.

WebAssembly binaries are obtained from source code like C/C++ or Rust [6]. The WebAssembly code is further interpreted or compiled ahead of time into machine code by engines such as the browsers. Since version 8, LLVM supports

Wasm as a backend opening the door for its vast collection of frontend languages. The LLVM support was encouraged by the seminal work of Zakai et al. with Emscripten. Emscripten is an open-source tool for compiling C/C++ to the WebAssembly. It uses LLVM to create Wasm, but it provides support for faster linking to the object files. Instead of all the IR being compiled by LLVM, the object file is prelinked with Wasm in a faster way.

WebAssembly for backend execution

Further browser context, the adoption of WebAssembly for backend has grown exponentially in the last four years. For instance, Cloudflare and Fastly adapted their platforms to provide FaaS directly with WebAssembly. In 2019, the bytecode alliance team [1] proposed WebAssembly System Interface (WASI). WASI is the foundation to build Wasm code outside of the browser with a system interface platform. It allows the adoption of WebAssembly outside web browsers [15] in heterogeneous platforms like the Edge or IoT [4, 13]. Previous studies resulted in performance increasing in terms of bandwidth saving, execution, and process-on-demand spawning [2, 10]. The words of Solomon Hykes [2], the former CEO of docker, show the impact of WASI:

> *If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!*

WebAssembly security and our motivation for diversification

WebAssembly is characterized by a robust security model [14]. It should run inside a sandboxed execution environment that provides protection against common security issues such as data corruption, code injection, and return-oriented programming (ROP). However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [19] or the binary itself [12]. Implementations in both browsers and standalone runtimes [4] are vulnerable. This means that if one environment is vulnerable, all the others are vulnerable in the same manner as the same WebAssembly binary is replicated, turning it into a monoculture problem.

On the other hand, the WebAssembly environment lacks natural diversity [33]. Compared to the work of Harrand et al. [?] , in WebAssembly, one could not use preexisting and different program versions to provide diversification for monoculture solving. In fact, according to the work of Hilbig et al. [6], the artificial variants created with one of our works contribute to the half of executable and available WebAssembly binaries in the wild.

[1] `https://bytecodealliance.org/`
[2] `https://twitter.com/solomonstre/status/1111004913222324225`

The current limitations on security and the lack of preexisting diversity motivate our work on software diversification as one possible mitigation among the wide range of security countermeasures.

2.2 Diversification and Superdiversification

Program diversification approaches can be applied at different stages of the development pipeline. This section analyzes the related works for both static and dynamic diversification. Besides, we motivate superoptimization strategies to provide a "superdiversifier" that uses intermediate solutions to search for optimal programs to provide program variants. Finally, we describe our contribution to the field.

Static diversification consists in synthesizing, building, and distributing different, functionally equivalent binaries to end-users. This aims at increasing the complexity and applicability of an attack against a large population of users [55]. Dealing with code-reuse attacks, Homescu et al. [35] propose inserting NOP instruction directly in LLVM IR to generate a variant with a different code layout at each compilation. In this area, Coppens et al. [36] use compiler transformations to iteratively diversify software. Their work aims to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach continuously applies a random selection of predefined transformations using a binary diffing tool as feedback [?] . A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them.

Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces required by a side-channel attack. Amarilli et al. [41] is the first to propose the generation of code variants against side-channel attacks. Agosta et al. [34] and Crane et al. [32] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software, while Couroussé et al. [27] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks.

Jackson et al. [39] have explored how to use NOP operations inserted during compiling time to statically diversify programs. Another idea is to use the optimization flags of several compilers to generate semantically equivalent binaries out of the same source code. These techniques place the compiler at the core of the diversification technique. However, this approach is limited by the number of available flags in the compiler implementation and because the optimization is applied in all possible places in the code at the same time.

Superoptimization

The search for optimal algorithms to compute a function is as older as of the first compiler. This problem is commonly solved by using human-written heuristics inside the compiler implementations. However, this solution has limitations. First, the optimizations are applied to small pieces of code and do not consider more complex processes like instruction selections, register allocation, and target-dependent optimizations. Second, the well-known phase ordering problem [38]. To solve this problem, Massalin et al. [56] proposed a superoptimizer, a statistical method to exhaustively explore all possible program constructions to find the smallest program. Given an input program, code superoptimization focuses on *searching* for a new program variant that is faster or smaller than the original code while preserving its correctness [28]. The search space for the optimal program is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. However, the exhaustive exploration approach becomes virtually impossible for larger instruction sets. Because of this, the paper proposes a pruning method over the search space and a fast probabilistic test to check programs' equivalence.

Apart from recent works in the area of Machine Learning [3], to the best of our knowledge, there are two main implementations for superoptimizers using two completely different strategies. Churchill et al. [25] implement STOKE to superoptimize large programs for the Google Native Client stack. They use a bounded verifier to ensure that every generated optimization goes through all the checks for semantic equivalence. STOKE uses a probabilistic approach, following a Monte-Carlo-Markov-Chain strategy to select code transformations that lead to smaller programs. On the other hand, Souper [49] automatically generates smaller programs for LLVM following an exhaustive enumerative synthesis. Souper finds subexpressions at the LLVM function level and builds all possible expressions from all the instructions that are no larger than the original subexpression. When Souper finds a replacement, it uses an SMT solver [45] to verify the semantic equivalence with the original program. Superoptimization is more expensive than traditional optimization heuristics in compilers yet, provides more profound and more robust code transformations.

Superdiversification and statement of novelty

While finding optimized code, the idea and the implementations of superoptimization discard intermediate solutions that are semantically equivalent to the original program. The discarding of intermediate solutions follows the principle of optimization, finding the best possible program. Jacob et al. [44] propose the use of a "superdiversification" technique, inspired by superoptimization, to synthesize individualized versions of programs, their main idea is to keep the intermediate

solutions finding the optimal program. The tool developed by Jacob et al. does not output only the optimal instruction sequence but any semantically equivalent sequences. Their work focuses on a specific subset of x86 instructions.

In this research, we contribute to the state of the art in artificially creating diversity. While the number of related work for software diversity is enormous, no approach has been applied to the context of WebAssembly. One of our contributions, CROW, extrapolates the idea of superdiversification for WebAssembly. CROW works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures, something not possible with the x86-specific approach of previous works. Furthermore, we conducted a sanity check for diversification preservation, researching to what extent browser compilers do not remove our introduced diversity.

CROW focuses on the static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a dynamic approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. It also can be used in fuzzing campaigns [?] to provide reliability. The diversification created by CROW can unleash hidden behaviors in compilers and interpreters. By generating several functionally equivalent and yet different variants, deeper bugs can be discovered. Thanks to CROW, a bug was discovered in the Lucet compiler [3]. Fastly acknowledged our work as part of a technical blog post [4] that describes the bug and the patch.

2.3 Runtime diversification

In this section, we highlight past works on runtime strategy for diversification. Besides, we describe and discuss the foundation that supports the composition of diverse yet semantically equivalent programs to enforce security. Finally, we describe our contribution to the field.

A randomization technique creates a set of unique executions for the very same program [52]. Seminal works include instruction-set randomization [51, 53] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker to ignore the key to inject executable code. This breaks the predictability of program execution and mitigates certain exploits.

Chew, and Song [54] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses), and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [52, 50] proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and

[3] `REPO`

[4] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates an extensive range of memory error exploits. Recent work in this field includes stack layout randomization against data-oriented programming [18], and memory safety violations [5], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [9].

Moving Target Defense and Multivariant execution

Moving Target Defense (MTD) for software was first proposed as a collection of techniques that aim to improve the security of a system by constantly moving its vulnerable components [8]. Usually, MTD techniques revolve around changing system inputs and configurations to reduce attack surfaces. This increases uncertainty for attackers and makes their attacks more difficult. Ultimately, potential attackers cannot hit what they cannot see. MTD can be implemented in different ways, including via dynamic runtime platforms [16]. Segupta et al. illustrated how a dynamic MTD system [23] can be applied to different technology stacks. Using this technique, the authors illustrated that some CVE related to specific database engines could be avoided.

On the same topic, Multivariant Execution (MVE) can be seen as a Moving Target Defense strategy. In 2006, security researchers at University of Virginia laid the foundations of a novel approach to security that consists in executing multiple variants of the same program. They called this "N-variant systems" [48]. Bruschi et al. [47] and Salamat et al. [46] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on Multivariant Execution (MVE) for mitigating memory vulnerabilities [20] and other specific security problems incl. return-oriented programming attacks [29] and code injection [40].A key design decision of MVE is whether it is achieved in kernel space [17], in user-space [42], with exploiting hardware features [26], or even through code polymorphism [22]. Finally, one can neatly exploit the limit case of executing only two variants [37**?** , 30]. Notably, Davi et al. proposed Isomeron [31], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy a potential attacker cannot predict whether the original or the variant of a program will execute.

Statement of novelty

Researching on MVE in a distributed setting like the Edge [?] has been less researched. Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [1] for the sake of efficiency. Since CROW offers static and runtime diversity for WebAssembly

and its adoption for the Edge and backend executions are becoming security-sensitive fields, we propose an original kind of MVE, MEWE. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE.

With MEWE, We contribute to the field of randomization at two stages. First, we automatically generate variants of a given program with CROW, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable behaviors at each run of the program.

2.4 Conclusions

Software Diversification has been widely researched, not being the case in the WebAssembly context. With this dissertation, we aim to settle down the foundation to study automatic diversification for WebAssembly. We contribute to the field of artificial diversity by extending the superdiversifier idea of Jacob et al. [44]. We empirically demonstrate that CROW provides robust program diversification. Finally, we propose a novel approach of merging program variants to provide multivariant execution. Our contributions are obtained by following the methodology described in Chapter 3.